Unifying Verification Paradigms (Extended Abstract)*

To appear in the Proceedings of FTRTFT'96

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. The field of formal methods is blessed with an overabundance of formalisms (functional, relational, automata-theoretic, modal, and temporal), techniques (resolution, rewriting, induction, and model checking), and application areas (hardware, reactive, fault-tolerant, real-time, and hybrid systems). No single verification approach has proven convincingly superior to the others. I argue that it is both necessary and desirable to develop a unified framework within which different approaches can coexist. The paper outlines some preliminary efforts in this direction in the context of SRI's PVS system. These efforts include the embedding of special-purpose formalisms (e.g., the Duration Calculus) into the general-purpose PVS logic, the integration of theorem proving with various forms of model checking, and the application of theorem proving and model checking to the analysis of tabular specifications.

1 Introduction

There has been a dramatic proliferation of verification formalisms in recent times. Volume B of the *Handbook of Theoretical Computer Science* [40] takes up the bulk of nearly 1300 pages to survey a small fraction of these formalisms (e.g.,

* Supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931 and CCR-930044. Some of the applications described were undertaken for NASA Langley Research Center under contracts NAS1-18969 and NAS1-20334 and for ARPA through NASA Ames Research Center under contract NASA-NAG-2-891. The views presented in this paper bear the influence of several of my colleagues including John Rushby, Sam Owre, Jens Skakkebæk, Judy Crow, David Cyrluk, Mandayam Srivas, Patrick Lincoln, and Klaus Havelund. Whether they or the funding agencies would concur with these views is another matter. The word *paradigm* is perhaps being misused in the title, but the author is not the first to perpetrate this abuse.

Hoare logics, modal and temporal logics, process algebras). Clearly, many of these formalisms are useful in obtaining a metatheoretic characterization of completeness, expressiveness, and decidability, but from a practical viewpoint, one has to wonder whether this cornucopia of formalisms really offers any significant advantage over conventional mathematics (e.g., predicate calculus, set theory, algebra, analysis, combinatorics). This multiplicity of formalisms raises several obvious questions and challenges:

- Who needs so many formalisms?
- Does any one formalism address a practical problem in its totality?
- How can we make it easier to implement a novel idea or formalism?
- How can we make different formalisms interact in a coherent manner?

In response to these questions, I believe that:

- Though ordinary mathematics is adequate for describing computational phenomena, formalism provides a convenient layer of abstraction along with notational and deductive tools. For example, temporal logic [28] can succinctly capture the fairness assumptions and the safety and liveness properties of finite-state programs. Furthermore, the propositional fragment of temporal logic is decidable. This fragment of temporal logic can also be used for model checking and for synthesizing control programs [6]. Similar arguments can be made for the numerous variants of Hoare logics, process algebras, algebraic datatype specification languages, dataflow languages, and others.
- In my experience, no individual special-purpose formalism fully addresses the verification problems of practical significance. Though formalism does make it convenient to use mathematics, it does not avoid the need for mathematics. For many practical problems, the main burden of verification is still in the sheer amount of conventional mathematics required to carry out a successful verification. For example, Dutertre [8] found it necessary to build a library of basic facts of mathematical analysis as a prelude to the formalization of hybrid systems. Other work in the verification of real-time and fault-tolerant systems [7] also exhibits a heavy dependence on basic mathematics that are not alleviated by the use of special-purpose formalisms.
- Logical frameworks like Isabelle [33] do make it easier to implement proof checkers for new formalisms. However, as we have already seen, individual formalisms are not adequate for practical verification. For verification purposes, we can obtain a more synergistic interaction between different formalisms by embedding these as theories and notational front-ends within a general-purpose framework for conventional mathematics. In this paper, we describe a few such attempts at integrating special-purpose formalisms and deductive procedures into the framework of PVS.
- Even if we can embed special-purpose formalisms within a general-purpose verification framework, very little has been achieved unless these formalisms can interact smoothly. For example, a unified approach should allow the use

of induction, rewriting, and propositional logic for the verification of hardware components, and the application of temporal logic or process algebra for verifying the asynchronous interaction between hardware components.

In an attempt to do something about these beliefs, we have been engaged in the development of a verification framework named PVS [30]. While PVS is a general-purpose verification system based on typed higher-order logic, it provides a notational framework for describing computational phenomena, and incorporates a variety of decision procedures for reasoning effectively about computation. PVS has been used to:

- Develop domain libraries for finite sets, cardinalities, bit-vectors, and reals [8].[2]
- Verify challenging exercises from areas such as hardware [39], fault-tolerant algorithms [27], real-time computing, and reactive systems [7, 19].
- Embed various formalisms like the Duration Calculus [37,38], branching and linear-time modal logics [34], action systems [25], and I/O automata [1].

This paper describes some of the steps taken in PVS towards a coherent integration of different verification paradigms, and the larger challenges that remain. The basic theme of this paper is that the effective mechanization of formal methods requires a careful and coherent integration of mathematical theories, deductive procedures, and special-purpose formalisms. Section 2 outlines a small PVS example showing that a fair amount of conventional mathematics is needed to formalize and verify even elementary computational problems. We then give three examples of integration of special-purpose formalisms and deductive techniques within the general-purpose framework provided by PVS. Section 3 describes the integration of symbolic model checking as a decision procedure in PVS. It shows how temporal program logics like CTL and can be defined in PVS, and model checking and theorem proving can be combined in the verification of an N-process mutual exclusion algorithm. Section 4 describes the construction of a Duration Calculus verification tool by means of a semantic embedding in PVS. The advantage of this approach is that deductive capabilities of PVS are combined with the notational convenience of the Duration Calculus. Section 5 outlines the extension of PVS with support for various kinds of tabular specification notations that are popular in software requirements engineering. The deductive and model checking capabilities of PVS are used in the analysis of such tabular specifications. Section 6 presents some observations and challenges.

2 The Need for Conventional Mathematics

This section describes a small PVS example showing that a significant amount of conventional mathematics is required to formalize even basic computational phenomena. This example is the well known Rabin-Scott result on the equivalence

[2] The PVS libraries and numerous PVS-related papers and reports are available from the URL: `http://www.csl.sri.com/pvs.html`.

of deterministic (DFA) and nondeterministic (NFA) versions of finite automata.[3] The following specification of a DFA happens to be in PVS but it is not that different from the corresponding textbook specification. It describes a theory with parameters `Sigma` denoting the alphabet type, `state` denoting the state space type, `start` identifying a distinguished start state, a transition function `delta` mapping each alphabet-state combination to the end state of the transition, and a set[4] `final?` of accepting or final states. The iterated transition function on a string of the alphabet `Sigma` is given by `DELTA`. The set of accepted strings is characterized by the predicate `DAccept?` which checks that the state returned by `DELTA` on the given string and the start state is a final state.

```
DFA  [Sigma : TYPE,
       state : TYPE,
       start : state,
       delta : [Sigma -> [state -> state]],
       final? : set[state] ]
    : THEORY
  BEGIN

   DELTA((string : list[Sigma]))((S : state)):
             RECURSIVE state =
     (CASES string OF
         null : S,
         cons(a, x): delta(a)(DELTA(x)(S))
      ENDCASES)
     MEASURE length(string)

   DAccept?((string : list[Sigma])) : bool =
       final?(DELTA(string)(start))
  END DFA
```

In this simple specification, we have already seen the use of sets (i.e., predicates like `final?`), curried higher-order operations (`delta` and `DELTA`), abstract datatypes (`list`), recursion, and arithmetic (for the termination of `DELTA`). The purpose of the curried form of the transition function `delta` will become apparent in the specification `NFA` below. The reason for using a parametric theory for `DFA` is so that useful definitions and facts about DFAs can be imported by suitably instantiating these parameters, as is done in the theory `equiv` below.

The specification of NFAs is similar. We first define the notion of a union or least upper bound `LUB(SS)` of a set of sets `SS` over the type `T`.

[3] This example was posed as a challenge by Debora Weber-Wulff.

[4] PVS is a typed higher-order logic where sets are just predicates over a specific type, rather than the richer notion given by, say, Zermelo-Fraenkel set theory.

```
LUB [T : TYPE]: THEORY
  BEGIN

    LUB((SS: set[set[T]]))(s : T) : bool =
        (EXISTS (S : set[T]): SS(S) AND S(s))
  END LUB
```

The key difference between the NFA and DFA theories is that the range of ndelta is now a set of states corresponding to a nondeterministic transition. The iterated transition function NDELTA returns a set of states by taking the union of the image[5] of the curried operation ndelta with respect to the set of states returned in the recursive invocation of NDELTA. A string is accepted according to Accept? if and only if the set of states returned by NDELTA on the string and the start state contains a final state.

```
NFA   [Sigma : TYPE,
       state : TYPE,
       start : state,
       ndelta : [Sigma -> [state -> set[state]]],
       final? : set[state] ]
    : THEORY
  BEGIN
    IMPORTING LUB[state]

    NDELTA((string : list[Sigma]))((s : state)) :
           RECURSIVE set[state] =
       (CASES string OF
         null : singleton(s),
         cons(a, x): LUB(image(ndelta(a), NDELTA(x)(s)))
        ENDCASES)
     MEASURE length(string)

   Accept?((string : list[Sigma])) : bool =
     (EXISTS (r : (final?)) :
       member(r, NDELTA(string)(start)))
  END NFA
```

Since every DFA is trivially an NFA, the equivalence between NFAs and DFAs is shown by constructing a DFA with the same language as a given NFA. This is done in the theory equiv which has the same parameterization as the NFA theory. The first declaration imports the corresponding NFA theory.

[5] The image operation is defined in the PVS prelude theories.

```
equiv[Sigma : TYPE,
      state : TYPE,
      start : state,
      ndelta : [Sigma -> [state -> set[state]]],
      final? : set[state] ]: THEORY

  BEGIN

   IMPORTING NFA[Sigma, state, start, ndelta, final?]
   ⋮
  END equiv
```

The corresponding `DFA` theory is imported by first constructing a `DFA` state type `dstate` as the type consisting of subsets of the `NFA` state type `state`. The corresponding deterministic transition function `delta` then maps a `dstate` to a `dstate` so as to correspond to an `NFA` transition. The final `DFA` states given by `dfinal?` are those subsets of `state` that contain an `NFA` final state. The `DFA` start state is the singleton set containing the `NFA` start state. We can now import the appropriate instance of `DFA`.

```
   dstate: TYPE = set[state]

   delta((symbol : Sigma))((S : dstate)): dstate =
        LUB(image(ndelta(symbol), S))

   dfinal?((S : dstate)) : bool =
     (EXISTS (r : (final?)) : member(r, S))

   dstart : dstate = singleton(start)

   IMPORTING DFA[Sigma, dstate, dstart, delta, dfinal?]
```

The equivalence between the iterated transition functions given by lemma `main` can be established by a completely trivial structural induction on the string parameter. The language equivalence given by `equiv` then follows by straightforward predicate calculus.

```
   main: LEMMA
    (FORALL (x : list[Sigma]), (s : state):
       NDELTA(x)(s) = DELTA(x)(singleton(s)))

   equiv: THEOREM
    (FORALL (string : list[Sigma]):
       Accept?(string) IFF DAccept?(string))
```

The equivalence between NFAs with and without epsilon transitions (i.e., transitions that do not consume a symbol from the string) is correspondingly trivial. It too involves such familiar ideas as inductive definitions, structural induction, definition expansion, and predicate calculus. The textbook proofs of this equivalence are unduly circuitous [21]. They also overlook the observation that these equivalences make no use of the finiteness of the `state` type. The main point is that a lot of basic mathematics (predicate calculus, set theory, lists, arithmetic, recursion) is needed to describe even a simple computational phenomenon. The mechanical proofs are trivial because of the tightly integrated mechanization of equational reasoning (i.e., congruence closure for propagating ground equality information and rewriting with respect to definitions and lemmas), predicate calculus, set manipulations, datatype simplifications, and structural induction over recursive datatypes.

3 Integrating Theorem Proving and Model Checking

This section descibes the embedding of the mu-calculus and the branching-time temporal logic CTL in PVS, and the integration of model checking as a deductive procedure for handle properties stated in this formalism. This integration makes it possible to apply model checking and theorem proving in conjunction to verification tasks where neither technique in isolation would be effective.

Verification by means of theorem proving establishes a property P of a system M by proving P from M, i.e., $M \vdash P$, whereas model checking [6] demonstrates that M satisfies P, i.e., $M \models P$. Typically in model checking, M is a finite Kripke model of a system consisting of states and transitions between states. The model M can be either in an *explicit* state representation as an assignment of values to variables, or in a symbolic form where sets of states are represented by a Boolean formula, and the transition relations are also represented by Boolean formulas. The property P is stated in a linear or branching time temporal logic. The model checker can then deduce whether the model M satisfies P by progressively annotating each state with subformulas of P that hold on it.

Model checking can be viewed as a decision procedure for the finite-state fragment of a more general theory of extremal fixpoints of monotone predicate transformers. This theory is based on Park's mu-calculus [32]. Emerson and Lei [9] showed how the mu-calculus can be used to embed a number of finite-state formalisms. They also gave an efficient algorithm for checking mu-calculus formulas. Burch, Clarke, Dill, McMillan, and Hwang [3] showed how Park's mu-calculus could be decided using a symbolic representation in terms of binary decision diagrams. They showed how several other model checking paradigms such as branching and linear-time model checking, LTL validity checking, simulation checking in process algebra, and language containment could be captured within the unified framework of the mu-calculus. In joint work with Rajan and Srivas [34], we showed how the mu-calculus could be defined in a higher-order logic as a way of combining the benefits of theorem proving and model checking. We give a brief outline of this integration.

Park's mu-calculus can be presented in a definitional way (i.e., without axioms) in higher order logic. The PVS theory `mucalculus` below takes a parameter `T` corrresponding to the state type of the computation system or Kripke model. We can then lift the implication ordering on formulas to predicates on states. A monotone predicate transformer is one that preserves this ordering relation as checked by `ismono`. A fixpoint of a predicate transformer is defined by the relation `isfix`.

```
mucalculus[T:TYPE]: THEORY
BEGIN
  s: VAR T
  p,p1,p2: VAR pred[T]
  predtt: TYPE = [pred[T]->pred[T]]
  pp: VAR predtt
  setofpred: VAR pred[pred[T]]

  IMPORTING orders[pred[T]]
  <=(p1,p2):bool = FORALL s: p1(s) IMPLIES p2(s)

  ismono(pp):bool =
    FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2)

  isfix (pp,p):bool = (pp(p) = p)
  ⋮
END mucalculus
```

The least fixpoint is defined as the greatest lower bound of the pre-fixpoints and this is used to define the μ operator.

```
  glb (setofpred):pred[T] =
    LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES  p(s))

  lfp (pp):pred[T] = glb({p | pp(p) <= p})

  mu(pp): pred[T] = lfp(pp)
```

Correspondingly, we can prove that the least upper bound of the post-fixpoints of a monotone predicate transformer yields the greatest fixpoint, which then defines the ν operator.

```
  lub (setofpred):pred[T] =
    LAMBDA s: EXISTS p: member(p,setofpred) AND p(s)

  gfp (pp):pred[T] = lub({p | p <= (pp(p))})

  nu (pp):pred[T] = gfp(pp)
```

We can informally write `mu(LAMBDA Z: P(Z))` as $\mu Z.P(Z)$. The fixpoint operators can be explained intuitively in terms of infinite disjunctions as follows. Let $\top$ be $\lambda z_1 \ldots, z_n$: **true** and $\bot$ be $\lambda z_1, \ldots, z_n$: **false**, then if P is $\cup$-continuous (i.e., $P[\cup_{i\in\omega} Z_i] = \cup_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $j < k$), and Q is $\cap$-continuous (i.e., $P[\cap_{i\in\omega} Z_i] = \cap_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $k < j$), then

$$(\mu Z.P[Z])(z_1, \ldots, z_n) = \bigvee_i P^i[\bot](z_1, \ldots, z_n)$$

$$(\nu Z.Q[Z])(z_1, \ldots, z_n) = \bigwedge_i Q^i[\top](z_1, \ldots, z_n)$$

Using PVS, we can prove that the `lfp` operator does actually return the least fixpoint, and consequently derive a useful least fixpoint induction principle. We can also prove various lemmas such as that μ and ν are duals, that $\cup$- or $\cap$-continuity implies monotonicity, and that for $\cup$-continuous predicate transformers P, the least fixpoint can be obtained by taking the least upper bound of iterates of P from $\bot$. None of these results requires the computation state space to be of bounded finite size. When the state type `T` is finite, the least and greatest fixpoints can be explicitly computed. In PVS, this is done by carrying out a binary encoding the variables of state type as an n-tuple of booleans, and using a package due to Janssen [24] which employs binary decision diagrams (BDDs) to symbolically compute least and greatest fixpoints over the booleans. With the ability to represent finite states and compute fixpoints, we can, for example, compute the reachability relation on the program graph and represent the operators of the branching-time temporal logic CTL.

The temporal operators of CTL are easily defined in terms of the mu-calculus. We give just three simple instances of these definitions. These operators are indexed by the next-state relation `N` of the computation. The temporal operator `EX` is defined so that `EX(N, p)` holds of a state if `p` holds of some next state. The temporal formula `EG(N, p)` holds of a state if there is an infinite computation path leading out where `p` always holds. The temporal formula `EU(N, p, q)` holds of a state if there is an infinite outgoing computation path where q eventually holds upto which point p holds. Note that the `connectives` theory overloads `AND`, `OR`, `NOT` to operate on predicates rather than booleans.

```
ctlops[state : TYPE]: THEORY
 BEGIN
  IMPORTING mucalculus[state], connectives[state]
  u,v,w: VAR state
  f,g,Q,P,p1,p2: VAR pred[state]
  Z: VAR pred[[state, state]]
  N: VAR [state, state->bool]

  EX(N,f)(u):bool =  (EXISTS v: (f(v) AND N(u, v)))

  EG(N,f):pred[state] = nu (LAMBDA Q: (f AND EX(N,Q)))

  EU(N,f,g):pred[state] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))
  ⋮
END ctlops
```

Fairness cannot be expressed in CTL but a fairness-enhanced CTL called fairCTL can be defined using the mu-calculus. Using fixpoint induction, we can prove a few useful theorems about CTL. The crucial theorem we have proved is a preservation theorem for `AG` properties that provides a sufficient condition for an abstraction mapping between two Kripke models to preserve the validity of `AG` properties. Using this theorem, we can establish the main safety property of an N-process mutual exclusion algorithm by applying a suitable abstraction to the induction step to reduce it to the correctness of a two-process protocol that is easily verified by means of model checking. Since the N-process algorithm is not a finite-state one, model checking alone will not be applicable here. Theorem proving could be used to verify the safety property, but this typically requires the discovery of a suitably strengthened invariant. The benefit of using model checking is that one does not need to discover a suitably strengthened inductive invariant. The discovery of the right abstraction mapping still requires some creativity but this is usually simpler than invariant discovery. In joint work with Havelund [12], we have applied abstraction to reduce the verification of a communication protocol to a form amenable to model checking.

With Rajan and Schuerman, we have also integrated model checking for the linear-time temporal logic LTL. Note that whereas branching-time properties are predicates on computation states, linear-time properties are predicates on computation traces. The integration of LTL model checking is done by translating LTL formulas into a Kripke automaton which is composed with the given Kripke model [5]. The LTL model checking problem then reduces to fairCTL model checking on this extended automaton. The integration of LTL is not as smooth as that of CTL since it requires a hand-crafted translation of the LTL satisfaction relation to that for fairCTL. However, the integration still retains the benefit allowing the combined use of theorem proving and model checking.

As already seen in Section 2, a fair amount of basic mathematics is needed to formalize the mu-calculus and CTL. Since the semantics of these formalisms has been embedded in PVS, we can combine the use of theorem proving and model checking within a single framework by formally decomposing problems into control-intensive parts where model checking is effective, and data-intensive parts where theorem proving techniques like rewriting and induction are needed. Such decompositions are crucial for the successful use of model checking.

4 Embedding the Duration Calculus

This section describes the embedding of a special-purpose formalism, the Duration Calculus (DC), within PVS. This embedding results in a system PC/DC where all the resources of PVS are exploited within the notational and deductive framework of the Duration Calculus. Since DC is used for reasoning about real-time systems, the arithmetic capabilities of PVS are especially useful for verifying DC properties. The embedding here of the Duration Calculus within PVS is not dissimilar to the semantic definition of the mu-calculus and CTL given in Section 3. The crucial difference here is that the proof theory of DC is developed and used to construct a proof checker for DC specifications and properties. Gordon [11] gives a good introduction to the embedding of program logics within higher-order logic.

The Duration Calculus [4] is a real-time interval temporal logic. A DC formula is a predicate over a time interval $[b, e]$, where $b \leq e$. Apart from the usual logical connectives and quantifiers, DC formulas can be constructed using a *chop* operator so that $A; B$ holds of $[b, e]$ if for some m, $b \leq m \leq e$, A holds of $[b, m]$ and B holds of $[m, e]$. The chop operator can be used to define the interval modalities $\Box$ (everywhere) and $\Diamond$ (somewhere). Atomic formulas in DC include equalities and arithmetic inequalities over DC terms. Terms in DC are constructed using arithmetic operations such as addition, subtraction, multiplication, and division. Terms are also evaluated over intervals. The main novelty in DC is the duration operator $\int s$, where s is a *state* formula, i.e., a predicate over the time parameter whose truth value varies only finitely often in any bounded time interval. On the interval $[b, e]$, $\int s$ represents the Riemann integral $\int_b^e s(t).dt$, that is it returns the accumulated time duration over the interval $[b, e]$ for which s has been true. In particular $\int \mathbf{1}$ is just the length of the interval $e - b$ and is represented by the term l.

DC can, for example, be used to prove a safety property of a gas burner that if gas is allowed to leak for no more than 1 second in a 30 second interval, then for any duration longer than 60 seconds, gas is leaking at most 1/20th of the time.

Skakkebæk [37,38], as part of his doctoral work, defined the semantics of DC within PVS, used the semantics to validate the inference rules of the Duration Calculus, and developed PVS strategies for applying the DC proof rules.[6] The resulting system PC/DC delivers the benefits of the arithmetic, equality, and

[6] The duration operator was actually defined axiomatically but the rest of the interval temporal logic was presented semantically.

typechecking capabilities of PVS within the language and notational framework of DC. Inal and Skakkebæk [23] have applied PC/DC to a several previously studied DC examples. They report finding errors in every example that they examined.

In the example of the gas burner, the requirements and design can be stated and typechecked in the DC dialect of PC/DC as shown below.

```
dcgasburner: DCTHEORY
BEGIN
  IMPORTING dcp_thms
  Leak: State

  Safe: IForm = n(20)*dur(Leak) <= len
  Req: IForm = len >= n(60) => Safe

  Des_1: IForm = []('Leak' => len <= n(1))
  Des_2: IForm = []((<>'Leak');(<>'not Leak');(<>'Leak') => len >= n(30))

  T: IForm = n(30)*dur(Leak) <= len
  Triple: IForm = T \/ (T;'Leak') \/ (T;'Leak';'not Leak')

  lem1: LEMMA Des_1 /\ Des_2 => Triple
  lem2: LEMMA Des_1 /\ Triple /\ (len >= n(60)) => Safe

  Gasburner: LEMMA (Des_1 /\ Des_2 /\ (len >= n(60))) => Safe
END dcgasburner
```

PC/DC proofs allow all the commands of PVS in addition to the DC inference rules. Subgoals in a PC/DC proof are also maintained in the DC dialect as shown in the example sequent below.

```
Preserve :

{-1}   (30 * dur(Leak) <= dur(tt));(30 * dur(Leak) <= dur(tt))
  |-------
{1}   30 * dur(Leak) <= dur(tt)
```

In addition to reinforcing the claims made in Sections 2 and 3, the above embedding shows how the deduction and typechecking capabilities can be exploited to support specification and verification in a special-purpose program logic. In this way, a useful general-purpose DC verification system could be constructed without the exorbitant overhead of building all the language and deduction components from scratch.

5 Tabular Specifications

We have already seen how special-purpose program logics and their associated deductive procedures can be integrated into PVS. This section presents the enhancement of PVS with support for tabular specification [31]. This enhancement yields a visually attractive specification notation that is supported by the specification and deductive capabilities already in PVS. In particular, PVS typechecking is used to generate well-formedness proof obligations. These proof obligations can be verified using the PVS proof checker. Furthermore, the model checking capabilities of PVS described in Section 3 can be used to verify and refine tabular specifications of mode transition systems.

A number of influential specification and verification methodologies are based on the use of tables [10, 17, 18, 26, 36]. There are a few special purpose tools that support the analysis of tabular specifications like SCR* [15, 16], RSML [13, 14], and Tablewise [20]. Since these tools are not attached to a general-purpose verification system, the analysis capabilities of these tools are quite limited.

A simple tabular specification can be given in terms of condition/entry pairs, as in the following specification of the sign of a number.

$$sign(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & +1 \\ \hline \end{array}$$

This table can be directly captured by means of the newly added PVS construct `COND` which captures conditional branching.

```
signs: TYPE = { x: int | x >= -1 & x <= 1}

x: VAR int

sign_cond(x): signs =
  COND
    x < 0  -> -1,
    x = 0  ->  0,
    x > 0  ->  1
  ENDCOND
```

Typechecking the above specification generates proof obligations requiring that the conditions of a `COND` construct be pairwise-disjoint and exhaustive. Internally, the PVS theorem prover treats the `COND` as a nested `IF-THEN-ELSE` so that the only special treatment for `COND` expressions is in the PVS typechecker. A visually more attractive specification of the `COND` expression above can be given using the newly added `TABLE` construct where the start of a condition, `->`, and `,` are replaced by `|`, `|`, and `||`, respectively. Note that the text following `%` in a line are comments that are placed to enhance readability.

```
sign_vtable(x): signs = TABLE
                %-------------%
                | x < 0 | -1 ||
                %-------------%
                | x = 0 |  0 ||
                %-------------%
                | x > 0 |  1 ||
                %-------------%
ENDTABLE
```

There are a few syntactic variants of the TABLE construct. When the conditions are enumerative, i.e., they are equalities of the form $z = a_i$ over subrange or enumerated type elements a_i, then the z parameter can be extracted out as a column heading.

```
modes: TYPE = { off, armed, engaged }

value(m:modes):bool = TABLE
         m
        %-----------------%
        | off     | false ||
        %-----------------%
        | armed   | true  ||
        %-----------------%
        | engaged | true  ||
        %-----------------%
ENDTABLE
```

Other variants of the TABLE construct allow the specification of horizontal 1-dimensional tables where conditions appear as column headings, 2-dimensional conditional or enumerative tables. Blank entries in tables are treated as unreachable and the corresponding proof obligations are generated by the PVS type checker.

The TABLE construct has been used to capture an SCR-style specification [2] of the mode-transition table of a simple cruise-controller where the modes (`off`, `inactive`, `cruise`, `override`) are controlled by the environment conditions (`ignited`, `running`, `toofast`, `brake`, `activate`, `deactivate`, `resume`). The table indicates the next mode corresponding to the current mode and the status of the environment conditions. The tabular specification generates well-formedness proof obligations that can be verified using the PVS proof checker. Deeper assurance regarding the correctness of the specification can be obtained by applying CTL model checking to check such properties as:

1. When the mode is `off`, the ignition is off (i.e., `ignited` is *false*).
2. In modes other than `off`, the ignition is on (i.e., `ignited` is *true*).
3. In `inactive` mode, either the engine is not `running` or the cruise control is not `activated`.
4. In `cruise` mode, the engine is `running`, the vehicle is not going `toofast`, the `brake` is not on, and `deactivate` is not selected.
5. In `override` mode, the engine is `running`.

Atlee and Gannon [2] used a hand-translation of the SCR specification into SMV [29] in order to perform model checking, whereas the tabular specification in PVS is directly combined with the PVS model checker.

Tabular specification and verification have also been applied to a toy autopilot specification where there are two interacting components: one component sets the climb angle, and the other sets the target altitude. We can use model checking to prove safety and liveness properties of a tabular specification of the auto-pilot system and also to demonstrate that one tabular specification implements another. PVS tables were used to capture the quotient lookup table in the verification of hardware for SRT division [35]. These example applications illustrate the value of integrating tabular notation into a general-purpose verification framework. Through this integration, we can exploit the visual appeal and simplicity of tables in system specification, and also employ sophisticated deductive tools in the analysis of tabular specifications.

6 Conclusions

I have argued that while special-purpose formalisms and verification tools are important, an attempt should be made to integrate them within a general-purpose framework based on conventional mathematics. I have given illustrative examples where symbolic model checking has been integrated into PVS as a decision procedure, the Duration Calculus has been embedded as a notational front-end to PVS, and PVS has been enhanced with support for tabular specification. In each case, the combination exhibited a useful synergy between generality and specificity.

What does all this have to do with fault-tolerant and real-time systems? PVS has been used in the verification of several fault-tolerant and real-time systems. These examples rely quite heavily on the conventional mathematical capabilities of PVS (e.g., equational and arithmetic decision procedures, induction, rewriting). The amount of mathematics can be quite overwhelming if it is not organized into theories, libraries, and special-purpose formalisms. A number of other users have similarly employed PVS to support formalisms that they previously employed by hand [1, 19, 25].

PVS, like other similar systems, was not explicitly designed with the anticipation of issues of integration and unification. It was in fact designed as a framework for developing conventional mathematics that integrated the use of decision procedures, rewriting, induction, and simplification with typechecking. The integrated nature of PVS has proved useful in the definition and integration of more specialized logics and deductive tools, but such integration work still has to be carried out in a somewhat *ad hoc* manner.

Logical frameworks like Isabelle [33] offer considerable convenience in defining and constructing deductive tools for specialized logics, but lack any systematic support for a coherent integration of the capabilities of different formalisms. The clear technical challenge is to design a system that offers the definitional convenience of logical frameworks together with coherent integration and interaction between different formalisms and deductive tools as demonstrated by the examples shown in this paper.

In the end, the basic difficulty of formal methods lies in the sheer scale of mathematics that is needed. Models, methods, tools, and notations provide useful navigational aids but these are not all that useful in isolation. Unification of the sort described in this paper is a necessary step towards overcoming the problem of scale.

References

1. M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, Boston MA, June 1996. IEEE Computer Society Press. To Appear.
2. Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
3. J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.
4. Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.
5. E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David Dill, editor, *Computer-Aided Verification 94*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Stanford, CA, June 1994. Springer Verlag.
6. E. M. Clarke and O. Grumberg. Research on automatic verication of finite state concurrent systems. In *Annual Review of Computer Science*, pages 269–290. Annual Reviews, Inc., 1987.
7. David Cyrluk, Patrick Lincoln, Steven Miller, Paliath Narendran, Sam Owre, Sreeranga Rajan, John Rushby, Natarajan Shankar, Jens Ulrik Skakkebæk, Mandayam Srivas, and Friedrich von Henke. Seven papers on mechanized formal verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1995.
8. Bruno Dutertre. Elements of mathematical analysis in PVS. In *International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Turku, Finland, August 1996. Springer-Verlag. To appear.
9. E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.
10. S. Faulk and P. Clements. The NRL Software Cost Reduction (SCR) requirements specification methodology. In *Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987. IEEE Computer Society.
11. Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 387–439, New York, NY, 1989. Springer-Verlag.
12. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

13. Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.
14. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
15. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMPASS [22], pages 109–122.
16. Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society.
17. K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.
18. K. L. Heninger et al. Software requirements for the A-7E aircraft. NRL Report 3876, Naval Research Laboratory, November 1978.
19. Jozef Hooman. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, Lübeck, Germany, September 1994. Springer-Verlag.
20. D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In COMPASS [22], pages 97–108.
21. John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.
22. *COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.
23. Recep Inal and Jens U. Skakkebæk. Applying a mechanized duration calculus assistant. In Hans Rischel, editor, *Nordic Seminar on Dependable Computing Systems*, pages 69–80, Lyngby, Denmark, August 1994. Technical University of Denmark.
24. G. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
25. Pertti Kellomäki. Mechanical verification of DisCo specifications. In *Israeli-Finnish Binational Symposium on Specification, Development, and Verification of Concurrent Systems*, Haifa, Israel, January 1996. The Technion.
26. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
27. Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.
28. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification*. Springer-Verlag, New York, NY, 1992.
29. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.
30. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

31. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html` .
32. David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 19.
33. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
34. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
35. H. Rueß, N. Shankar, and M.K. Srivas. Modular verification of SRT division. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.
36. Lance Sherry. A structured approach to requirements specification for software-based systems using operational procedures. In *13th AIAA/IEEE Digital Avionics Systems Conference*, pages 64–69, Phoenix, AZ, October 1994.
37. Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
38. Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic.* PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, November 1994.
39. Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.
40. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.